

Re: Two Click disassembly/reassembly

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2006-01/msg01418.html>

- *From:* "randyhyde@xxxxxxxxxxxxxx" <randyhyde@xxxxxxxxxxxxxx>
 - *Date:* 24 Jan 2006 14:54:05 -0800
-

Alex McDonald wrote:

>>
>> No, it looks like a **compiler** rather than an **interpreter** (emulator).
>> And that means it will run about 2-10x faster than an emulator.
>
> OK, I see your point. However, there are some tasks that even a
> compiler can't handle here. This;
>
> ADD EAX, [EBX+10]
>
> (in whatever x86 notation takes your fancy) would be better emulated on
> a VM on the target processor than make the attempt to translate each
> and every statement into a set of equivalent opcodes on the target. For
> instance, using memory instead of registers, or emulating the stack
> runs the problem of state; what state have I left at the last
> instruction I generated code for? The above code leaves states of
> overflow, zero, carry to name but a few. Now stick a label on it, and
> jump to it from somewhere else. Because we need to retain states at run
> time (we can't know them all at compile time), we're emulating, not
> compiling.

This is where the optimizer comes in. The optimizer does a data flow analysis of the code to determine which of these flags will actually get used (or which flags it cannot determine do **not** get used) along some future control path, and then emits code to maintain **only** the necessary flags. Yep, it can be expensive on certain processors that have no concept of a carry or overflow flag. But you're going to have the same problem with emulation/interpretation.

>
> It's worse with statements like
>
> CALL \$+5
> POP EAX
>
> That leaves the address of the POP in EAX. But there are architectures
> where the IP must be on a 4byte boundary and where branch delay slots

Re: Two Click disassembly/reassembly

- > are required to change the IP; like the MIPS. I'd like to see
- > line-by-line compilation of the equivalent into MIPS; it's a serious
- > challenge.

Sure it's a serious challenge. But you're talking about optimization issues. That is, let's translate the *semantics* of the original code to get the best possible MIPS code. Simply translating the x86 code to something that will execute, albeit slowly, on the MIPS isn't the problem. The problem is the *slow* code you'll get if you do a trivial translation of the above to something that will work, though not efficiently, on the MIPS.

Again, the trivial stuff even Betov could do. But the code would run so slowly on something like the MIPS (a popular Pocket PC processor), particularly at the less than GHz speeds of MIPS processors used in Pocket PCs, that no one would use the translator. The real work is developing a decent optimizer. Apple (68K->PPC and Rosetta), Intel (IA64), and Transmeta have experience with this. Rene doesn't even understand the problems.

- > There is a possible one opcode alternative, as the MIPS has
- > a single instruction (jal) that leaves the return address /plus eight/
- > (after the branch delay slot) in a register. But that needs lookahead
- > to identify that the target of the CALL is a POP; now it's a compiler
- > again, rather than a line by line assembler. And what if the POP was
- > reached from a CALL [EBX] miles away in the code? This is seriously
- > hard stuff, even for a very smart compiler, and territory best handled
- > by an emulator.

Well, it's most certainly handled *easiest* (from the programmer's point of view) by an interpreter. But *good* optimization strategies *do* exist. Though the MIPS is much cruder than the PPC, the PPC has similar issues. Ugly code because opcodes are limited to 32 bits. But the trick is to simply use more instructions and figure out how to optimize them so that "more" doesn't turn into "a whole lot more". Keeping in mind that many modern RISC processors actually run slower, in general, than an x86, we don't really want to lose a whole lot of performance when translating from the x86 to something else.

- >
- >>
- >>> and to be quite honest, it would be easier
- >>> to write one than your mythical "encoder".
- >>
- >> Certainly writing an interpreter is far easier than writing a compiler.
- >> But the interpreter is *much* slower.
- >
- > And Betov's x86 source/MIPS backend assembler (as an example, and if
- > possible) would be as slow, if not slower still.

Re: Two Click disassembly/reassembly

You're assuming it even works. Given the state of his disassembler, which he claims is "finished", I'd give it about a 1% chance of working reasonably well. (At least the problem of cross compiling from one machine code to another is a possible and tractible, though conceptually difficult, problem :-)).

>>
>> In simple terms, he is describing an over-glorified macro processor
>> that translates each x86 assembly instruction into some comparable
>> sequence on a CPU found on a pocket PC or similar system. Of course,
>> the concept of optimization has never occurred to him, so the code he
>> would generate would be absolutely terrible.
>
> I would contend; not possible. There are too many variables to handle
> with states and side effects. That's why compiled languages deliberately
> remove them.

Difficult and inefficient, but certainly possible. I mean, it's quite easy to come up with a MIPS instruction sequence that faithfully reproduces the semantics of each x86 machine instruction. The only problem is that those sequences are quite long and you wouldn't want to emit them for each and every x86 instruction. Then again, maintaining such state probably wouldn't even occur to Rene until after about three years into the project, at which point he'd just announce that the project is "complete" and does a "two click assembly to MIPS (or whatever) code" and "Gee, isn't RosAsm great because it does this?" The fact that it wouldn't really be working except for some trivial demo cases wouldn't mean much to him.

>
>>
>> Rene reminds me of this Canadian guy, Roedy Green, from the 1990s
>> (though Roedy was a heck of a lot nicer). He used to be a *big*
>> supporter of assembly language. Everything was to be written in
>> assembly. One of the biggest supporters of assembly at the time. Then,
>> one day, he switched to Forth because of assembly's "limitations".
>> Someday, I expect the same sort of thing from Rene (probably when the
>> ReactOS team calls it quits).
>
> Please, let it be not Forth. I am an amateur Forth programmer, a keen
> advocate and maintainer of a Forth compiler (part public domain, part
> GPL). I couldn't bear the thought of Rene lecturing the "sub sh*ts" on
> the "one true way" over on comp.lang.forth. Or of wannabee following
> his prophet and dribbling incoherent replies to every post there.

:-)
Cheers,
Randy Hyde

.

• **References:**

- ◆ **Two Click disassembly/reassembly**
◇ From: randyhyde@xxxxxxxxxxxxxx
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Frank Kotler
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Evenbit
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Frank Kotler
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Evenbit
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Frank Kotler
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Betov
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: santosh
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Betov
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: santosh
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Betov
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Alex McDonald
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: randyhyde@xxxxxxxxxxxxxx
- ◆ **Re: Two Click disassembly/reassembly**
◇ From: Alex McDonald

- Prev by Date: **Re: Two Click disassembly/reassembly**
- Next by Date: **Pecker Contests**
- Previous by thread: **Re: Two Click disassembly/reassembly**
- Next by thread: **Re: Two Click disassembly/reassembly**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**